Ville Holopainen

Jatkuvan integraation laadun varmistus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Automaatiotekniikka

Insinöörityö

23.12.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Ville Holopainen Jatkuvan integraation laadun varmistus 31 sivua 23.12.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Automaatiotekniikka
Suuntautumisvaihtoehto	Automaation tietotekniikka
Ohjaaja(t)	Lehtori Markku Inkinen T&K Päällikkö, Työasemaohjelmisto Ilkka Haukkavaara

Insinöörityön tavoitteena oli parantaa tuotteen ohjelmistokehityksen jatkuvan integraation (continuous integration) laadunvarmistusta. Lisättäviä laatua parantavia menetelmiä olivat ohjelmistotuotteen suorituskykymittaus, yksikkötestien suorituksen kattavuusmittaus sekä ohjelmistokoodin staattinen analyysi. Osana työtä toteutettiin laatumenetelmien tulosten analysointi ja raportointi automatisoitiin tuotteen laadun varmistamiseksi.

Työn teoriaosuudessa tutustuttiin yleisellä tasolla ketterään ohjelmistokehitykseen, jonka keskeisenä osana on jatkuva integraatio. Työssä perehdytään syvemmin jatkuvan integraation laatua parantaviin menetelmiin. Työn toteutusosassa kerrotaan uusien mittausten ja analyysin toteutuksesta, työkalujen valinnasta ja raportoinnista.

Työn tuloksena kaikki tavoitteen mukaiset tehtävät on kehitetty ja lisätty jatkuvan integraation palvelimelle suoritettavaksi. Suorituskykymittauksen tarjoaman statistiikan perusteella on tehty jo parannuksia. Yksikkötestauksen kattavuusmittaus auttaa jatkossa parantamaan sekä laajentamaan testausta, jonka laadusta vastaa staattinen analyysi.

Avainsanat	Jatkuva integraatio, laadunvarmistus, ohjelmistokehitys, ohjelmistotuotanto

Author(s) Title Number of Pages Date	Ville Holopainen Quality assurance of continuous integration 31 pages 23 December 2014
Degree	Bachelor of Engineering
Degree Programme	Automation Technology
Specialisation option	Information technology in automation
Instructor(s)	Markku Inkinen, Senior Lecturer Ilkka Haukkavaara, R&D Manager, Workstation software

The target of this Bachelor's Thesis was to improve software product development quality assurance of continuous integration. Improving product quality methods to be added were software product performance measurement, unit test coverage measurement and static analysis of source code. Quality method result analysis and reporting was implemented as part of this thesis work.

Agile software development including continuous integration is presented in the theoretical section of this thesis. Improving product quality methods of continuous integration were studied further for more detailed level of information. Implementation, choosing the tools and reporting of new measurements and analysis is provided in the practical section of this thesis.

As a result of this project all methods based to requirements have been developed and added into continuous integration server to be performed. Improvements have already been generated based on statistics of performance measurement. The unit test coverage measurement will help to improve and expand the testing in future, and responsible for the quality is static analysis.

Keywords	Continuous integration, quality assurance, software development, software testing

Sisällys

Lyhenteet

CI	*Continuous integration*. Prosessi, jota käytetään ohjelmistokehityksessä.
COM	*Component Object Model*. Rajapinta ohjelmistokomponenteille
Exe	*Executable*. Ohjelman ajettavan tiedoston pääte.
FDA	*Food and Drug Administration*. Yhdysvaltain elintarvike- ja lääkevirasto.
IEC62304	*Medical device software – Software life cycle prosesses*. Ohjelmiston elinkaaristandardi.
NAS	*Network Access Storage*. Verkkolevy.
Scrum	Projektihallinnan viitekehys.
STL	*Standard Template Library*. C++-ohjelmointikielessä käytetty kirjasto, joka tarjoaa ohjelmoijalle tietorakenteita.
TDD	*Test-driven development*. Projektihallinnan viitekehys.
XP	*Extreme Programming*. Projektihallinnan viitekehys.

1 Johdanto

Tässä insinöörityössä käydään läpi jatkuvan integraation toiminnallisuutta. Työn alussa selvitetään, mitä jatkuva integraatio on, kenelle se on tarkoitettu sekä mitkä ovat sen hyvät ja huonot puolet. Työn loppuosassa käydään läpi työn osana jatkokehitettyä jatkuvan integraatiojärjestelmän toiminnallisuutta, jota käytetään CliniView-ohjelmistotuotteen kehityksessä.

CliniView on osa lääketieteellistä hammaskuvantamisjärjestelmää. Cliniview:n päätoiminnallisuudet ovat potilastietojen hallinta, hammasröntgenkuvien vastaanotto röntgenlaitteelta ja kuvien arkistointi.

Se lisäksi, että ketterät menetelmät ovat vakiinnuttaneet asemansa ohjelmistokehitysteollisuudessa, on uusien menetelmien tarve tunnistettu myös terveydenhuollon tuotekehitysvaatimuksissa. Muun muassa terveydenhuollon kansainväliset standardit ja asetukset, kuten IEC62304 ja FDA, edellyttävät perinteisen systeemitestauksen lisäksi kehittyneitä yksikkötestauksen menetelmiä sekä analysointityökalujen käyttöä.

2 Yritysesittely

PaloDEx Group Oy toimii kansainvälisillä terveydenhuollon markkinoilla keskittyen röntgentekniikka sovelluksiin, joita käytettyjen hammaskuvantamislaitteissa. PaloDEx Group Oy tarjoaa laadukkaita kuvantamisen ratkaisuja, jotka auttavat tekemään oikeita hoitopäätöksiä ja parantaa hammashoidon tasoa.

PaloDEx on maailman johtava digitaalisten ja analogisten hammaskuvantamislaitteiden sekä sisäisten ja suun ulkopuolisten sovelluksien suunnittelussa ja tuotannossa. Lisäksi yritys tarjoaa innovatiivisia ohjelmistoja, joiden avulla kuvan hallintaominaisuudet ovat erinomaiset. Yritys on edelläkävijänä tuonut ensimmäisen panoraamaröntgenlaitteiston markkinoille vuonna 1961, sekä lasketun radiografian (CR) epäsuoran digitaalisen kuvan talteenoton hammashoidossa vuodesta 1994.

3 Ohjelmiston tuotantotyö

3.1 Ketterä ohjelmistokehitys

Ketterä ohjelmistokehitys tarkoittaa menetelmää, jota käytetään ohjelmistotuotantoprojekteissa. Näille menetelmille on yhteistä toimivan ohjelmiston ensisijaisuus, suora viestintä ja nopea muutoksiin reagointi. Ketteriä menetelmiä on useita, muun muassa Extreme Programming (XP), Scrum, Test-driven development (TDD) ja Feature-driven development (FDD). Ketterät menetelmät pyrkivät pienentämään riskejä jakamalla ohjelmistoprojektin lyhyempiin jaksoihin. Jaksoon lisätään tehtäviä, jotka sisältävät projektisuunnittelun, vaatimusanalyysin, ohjelmistosuunnittelun, koodauksen, testauksen ja dokumentoinnin. Ketterissä menetelmissä pyritään saamaan julkaisukelpoinen ohjelmisto jokaisen jakson päätteeksi. Jakson aikana tehtävät pienet toteutustehtävät suunnitellaan tarkasti, sekä suunnitelmia voidaan päivittää herkemmin verrattuna perinteisiin menetelmiin. Jatkuva yhteistyö, joka mieluiten tapahtuu kasvokkain eri osapuolien kanssa, muokkaa suunnitelmia sen mukaan, mikä todetaan parhaimmaksi. Osapuoliin kuuluvat ohjelmoijien lisäksi tuotepäälliköt, loppukäyttäjät, testaajat ja muut sidosryhmät. [1]

Ketterät menetelmät nojautuvat 12 toimintaperiaatteeseen. [2]

- Tärkein tavoitteemme on tyydyttää asiakas toimittamalla tämän tarpeet täyttäviä versioita ohjelmistosta aikaisessa vaiheessa ja säännöllisesti.
- Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailukyvyn edistämiseksi.
- Toimitamme versioita toimivasta ohjelmistosta säännöllisesti, parin viikon tai kuukauden välein, ja suosimme lyhyempää aikaväliä.
- Liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan.
- Rakennamme projektit motivoituneiden yksilöiden ympärille.
- Annamme heille puitteet ja tuen, jonka he tarvitsevat ja luotamme siihen, että he saavat työn tehtyä.
- Tehokkain ja toimivin tapa tiedon välittämiseksi kehitystiimille ja tiimin jäsenten kesken on kasvokkain käytävä keskustelu.
- Toimiva ohjelmisto on edistymisen ensisijainen mittari.

- Ketterät menetelmät kannustavat kestävään toimintatapaan.
- Hankkeen omistajien, kehittäjien ja ohjelmiston käyttäjien tulisi pystyä ylläpitämään työtahtinsa hamaan tulevaisuuteen.
- Teknisen laadun ja ohjelmiston hyvän rakenteen jatkuva huomiointi edesauttaa ketteryyttä.
- Yksinkertaisuus - tekemättä jätettävän työn maksimointi - on oleellista.
- Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseorganisoituvissa tiimeissä.
- Tiimi tarkastelee säännöllisesti, kuinka parantaa tehokkuuttaan, ja mukauttaa toimintaansa sen mukaisesti.

Jatkuva integraatio on kehitetty tukemaan ketteriä menetelmiä. Toimittaaksemme version toimivasta ohjelmistosta säännöllisesti parin viikon tai kuukauden välein, on sen rakentaminen tehtävä automaattiseksi. Jatkuvan integraation palvelin hoitaa käännöksen sekä asennuspaketin luomisen. Jatkuva integraatio sisältää myös automatisoidun yksikkötestauksen ja sovelluksen käyttöliittymä testauksen puhtaassa ympäristössä. Tuotteen teknistä laatua voidaan parantaa lähdekoodin analysoinnilla. Tämänlaisia menetelmiä ovat sekä dynaaminen että staattinen analyysi. Jatkuvan integraation järjestelmän automatisoitu testaus ja analysointi raportoivat tulokset kehittäjille, jotta voidaan varmistua uuden version osien toimivuudesta.

3.2 Scrum

Scrum on projektihallinnan viitekehys, jota käytetään ketterässä ohjelmistokehityksessä. Scrumissa työskennellään toistavasti (iterative) ja lisäävästi (incremental), jolloin tuote kehittyy osittain valmiimmaksi useiden kehitysjaksojen aikana. Yhtä kehitysjaksoa kutsutaan sprintiksi.

Sprint kestää yleensä 1 - 4 viikkoa. Sprintin suunnittelupalaverissa sovitaan sisältö tulevaa sprinttiä varten, ja toteutettaviksi valitaan sellaisia tuotteen kehitysjonon (backlog) asioita (user story), joilla on sillä hetkellä suurin merkitys projektin onnistumiselle. Jokaisen sprintin lopuksi järjestetään katselmointi (demonstration), jossa kehitystiimi esittelee sprintissä valmistuneen sisällön sidosryhmille. Katselmoinnin lisäksi järjestetään myös retrospektiivi (retrospective), jossa tarkastellaan prosessin näkökulmasta mikä

sprintin aikana sujui hyvin ja mitä voitaisiin parantaa, mitä kannattaisi ottaa käyttöön ja mitä pitäisi välttää seuraavassa sprintissä. [3]

3.2.1 Roolit

Scrum-tiimi koostuu tuoteomistajasta (product owner), scrum-mestarista (scrum master) ja kehitystiimistä, jotka päättävät kunkin sprintin tavoitteet ja tehtävät sekä vastaavat siitä, että asetettuihin tavoitteisiin päästään.

Tuoteomistaja vastaa tuotteen arvon ja kehitystiimin työn arvon maksimoimisesta. Käytännössä tämä tapahtuu tuotteen vaatimuksien määrityksellä sekä järjestämällä tuotteen kehitysjonon sisällön scrum-tiimin kanssa yhteistyötä tehden. Tuoteomistaja tuntee tuotteen liiketoiminnan, edustaa asiakkaita ja käyttäjiä sekä varmistaa, että scrum-tiimi toteuttaa jokaisessa sprintissä sellaisia vaatimuksia, jotka ovat tuotteen onnistumisen kannalta keskeisimpiä. Vaatimuksien määrittelijänä tuoteomistaja hyväksyy katselmoinnissa edellisen sprintin tuoteversion. Tuoteomistaja myös auttaa kehitystiimiä ymmärtämään vaatimuksia ja tarvittaessa tarkentaa sprinttiin valittuja vaatimuksia toteutuksen aikana.

Scrum-mestarin tehtävänä on huolehtia siitä, että tiimi voi tehdä työtään optimaalisella tavalla sekä vastaa siitä, että kaikki ymmärtävät ja käyttävät Scrumia. Tämän lisäksi hän johtaa päivittäiset aamupalaverit. Hän tarkkailee työn etenemistä, ja jos sprintin tavoitteiden saavuttaminen alkaa näyttää epätodennäköiseltä, hän kommunikoi kehitystiimin ja tuoteomistajan kanssa korjaavista toimenpiteistä tilanteen parantamiseksi tai sprintin sisällön kaventamiseksi. Scrum-mestari myös suojaa tiimiä ulkopuoliselta häiriöltä, kuten sprintin aikana pyydetyiltä uusilta vaatimuksilta, ja tekee kaikkensa turvatakseen tiimilleen työrauhan.

Kehitystiimi koostuu ammattilaisista, joilla on tarvittava osaaminen muuttaa tuotteen kehitysjonon sisällön julkaisukelpoiseksi tuoteversioksi jokaisessa sprintissä. Kehitystiimi rakentaa yhdessä tuotteen. Tällä halutaan korostaa kunkin tiimiläisen olevan projektin kannalta yhtä tärkeä ja että tiimi yhdessä vastaa tuotteen kaikista puolista, ei koskaan yksittäinen henkilö. [3]

3.2.2 Prosessi

Ennen tuoteprojektin aloittamista luodaan korkean tason visio siitä, mitä projektilta halutaan. Visio vastaa kysymyksiin, miksi projekti tehdään ja mitä projektin lopputuotteena saadaan. Vision avulla voidaan aloittaa tuotteen työlistan muodostaminen.

Tuotteen työlistaan lisätään asioita, mitä tuotteessa saatetaan tarvita. Työlistasta vastaa tuotepäällikkö. Vastuu sisältää tuotteen sisällön, saatavuuden sekä priorisoinnin. Tuotteen työlista ei ole koskaan valmis, vaan se kehittyy, kun tuote sekä ympäristö jossa sitä tullaan käyttämään kehittyvät. Julkaisusuunnitelmaksi kutsutaan usein sellaista tuotteen kehitysjonoa, joka sisältää kaikki tuotteen työlistasta seuraavaan julkaisuun valitut kohdat.

Sprintin suunnittelupalaverissa siirretään tuotteen työlistalta valittuja toiminnallisuuksia, user storyja, sprintin työlistaan. Sprintin aikana vaatimusten muuttaminen on kiellettyä, ja tiimillä on täysi vapaus tehdä tarpeelliseksi katsomiaan toimenpiteitä, jotta sovittu sprintin päämäärä voidaan saavuttaa. Tiimi organisoi itsensä parhaaksi katsomallaan tavalla. Sprintin aikana toteutumista voi seurata edistymiskäyrän (sprint burndown chart) avulla, joka kuvaa jäljellä olevaa työtä ajan funktiona.

Joka päivä pidetään tilannekatsaus (daily standup meeting), johon osallistuu vähintään kehitystiimi ja scrum-mestari, jossa kukin tiimin jäsen vastaa kolmeen kysymykseen:

- Mitä teit edellisen päiväpalaverin jälkeen?
- Mitä aiot tehdä seuraavan 24 tunnin aikana?
- Mitkä tekijät estävät (tai hidastavat) sinua etenemästä työssä?

Palaveri on myös avoin kaikille muille, jotka ovat jollain tavalla projektista kiinnostuneita. Vain tiimiläiset saavat kuitenkin puhua. Tapahtuman tarkoitus on nimenomaisesti tarjota kaikille tieto siitä, miten projekti etenee ja mitä ongelmia sillä on. Jos jostain asiasta pitää keskustella tarkemmin, sitä varten pidetään oma palaverinsa, jossa ovat läsnä vain ne henkilöt, joita asia koskee.

Jokaisen sprintin lopuksi sprintin katselmoinnissa tiimi esittelee valmistunutta tuotetta tuotteen omistajalle ja sopeutetaan tarvittaessa tuotteen kehitysjonoa. Sprintin lopuksi tuotteen siis pitäisi olla periaatteessa käyttöönotettavissa: Se on toteutettu, testattu,

dokumentoitu, käyttöliittymä on valmis ja niin edelleen. Näin omistaja voi päättää joko seuraavan sprintin tekemisestä tai tuotteen käyttöönottamisesta. Sprintin katselmointiin saa osallistua kuka tahansa, joka on projektista kiinnostunut.

Katselmoinnin lisäksi sprintin lopuksi tiimi, scrum-mestari ja omistaja tarkastelevat päättynyttä sprinttiä. Tapahtumaa kutsutaan retrospektiiviksi. Sprintin retrospektiivi pidetään sprinttikatselmuksen jälkeen ja ennen seuraavan sprintin suunnittelupalaveria. Jokainen tiimin jäsen kertoo omasta näkökulmastaan, mikä sprintissä meni hyvin, mitä voisi parantaa ja mitä pitäisi välttää. Lopuksi tiimi yhdessä priorisoi kehityskohteet ja pyrkii toteuttamaan muutokset seuraavan sprintin aikana. [3]

4 Jatkuva integraatio

Jatkuva integraatio on prosessi, jota käytetään ohjelmistokehityksessä. Termi integraatio tarkoittaa yhden kehittäjän lähdekoodiin tekemien muutosten liittämistä muiden kehittäjien työkopioon versionhallinnan sekä jatkuvan integraation palvelimen, CI-palvelin, kautta. Jatkuvuutta kuvaa yleensä useasti päivässä tapahtuva integraatio.

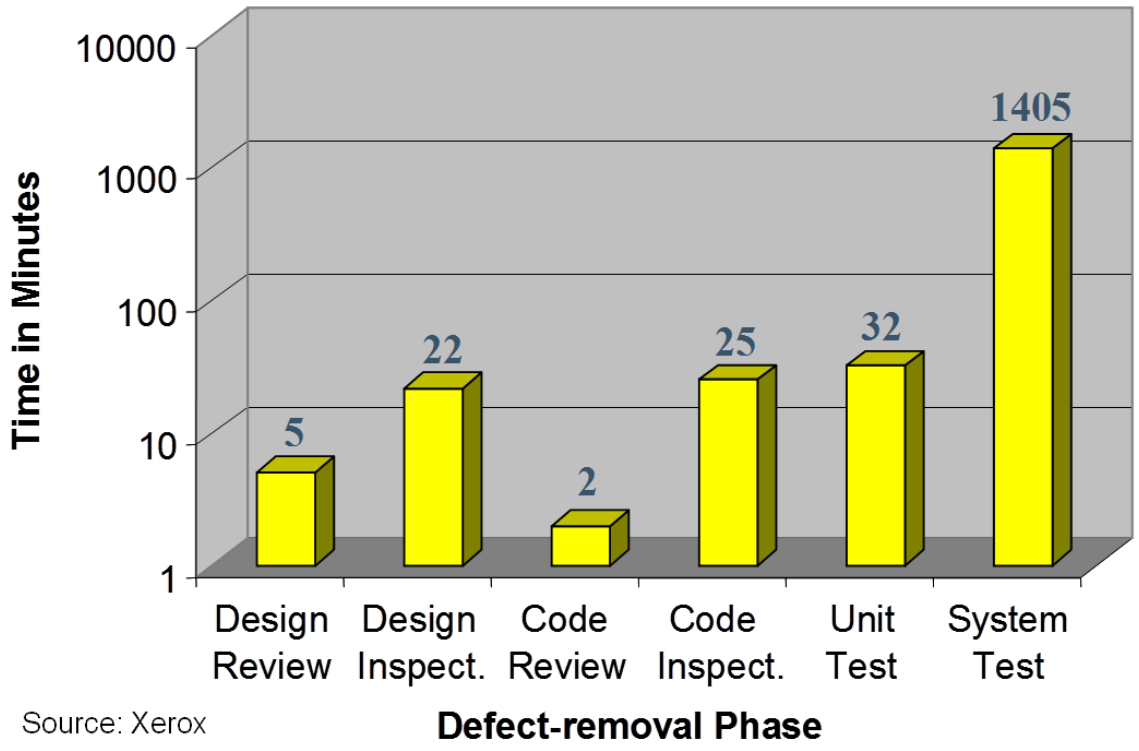
Versionhallinnasta löytyy ohjelmiston kehittäjille jaettu päälinja, minne kehittäjä liittää omaan työkopioon kehitetyt muutokset. Kun päälinjaan lisätään muutos, CI-palvelin havaitsee sen ja käynnistää toiminnan. CI-palvelin päivittää ensin oman työkopionsa versionhallinnasta, suorittaa tarvittavat rakennustoimenpiteet (build), mittaa lähdekoodin metriikan sekä ajaa yksikkö- ja integraatiotestit. Build tuottaa myös ajettavan version järjestelmästä, joka asennetaan puhtaaseen ympäristöön automaattista testausta varten. [4]

4.1.1 Käännökset

Ennen jatkuvaa integraatiota jokainen lähdekoodin lisäys ja muutos piti integroida manuaalisesti. Mitä pidempään muutosta teki, sitä varmemmin alkuperäiseen lähdekoodiin oli tullut jo muutoksia kun oma muutos piti lisätä. Tämä tarkoittaa integraatiota uuden koodin kanssa ja pahimmassa tapauksessa koko muutos oli uudelleen kirjoitettava.

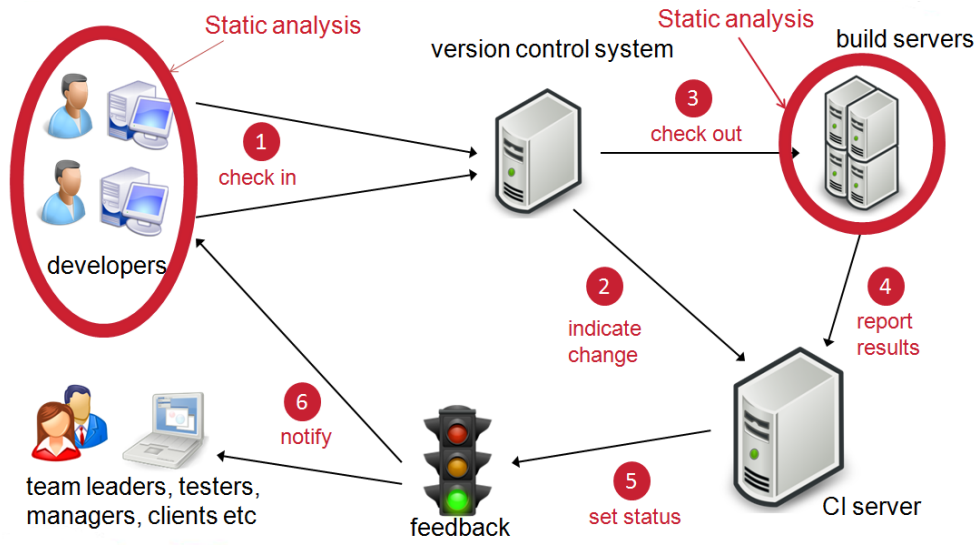
Integraatio-ongelmien korjaukset vievät sitä enemmän aikaa, mitä myöhäisemmässä vaiheessa ne löytyvät kuvan 1 mukaisesti. Mikäli virhe löytyy jo suunnitteluvaiheessa,

on se helppo korjata suunnitelmaan. Jos virhe löytyy yksikkötestauksessa, voi virhe vaikuttaa myös alkuperäiseen suunnitelmaan ja tämän myötä molemmat on korjattava, ja sen jälkeen toteutettava uudelleen. Jatkuvan integraation tarkoituksena on poistaa pitkään hautuneet muutokset ja integroida nimensä mukaan jatkuvasti muiden muutokset omaan lähdekoodiin, että integraatio-ongelmat pysyisivät pienempinä.



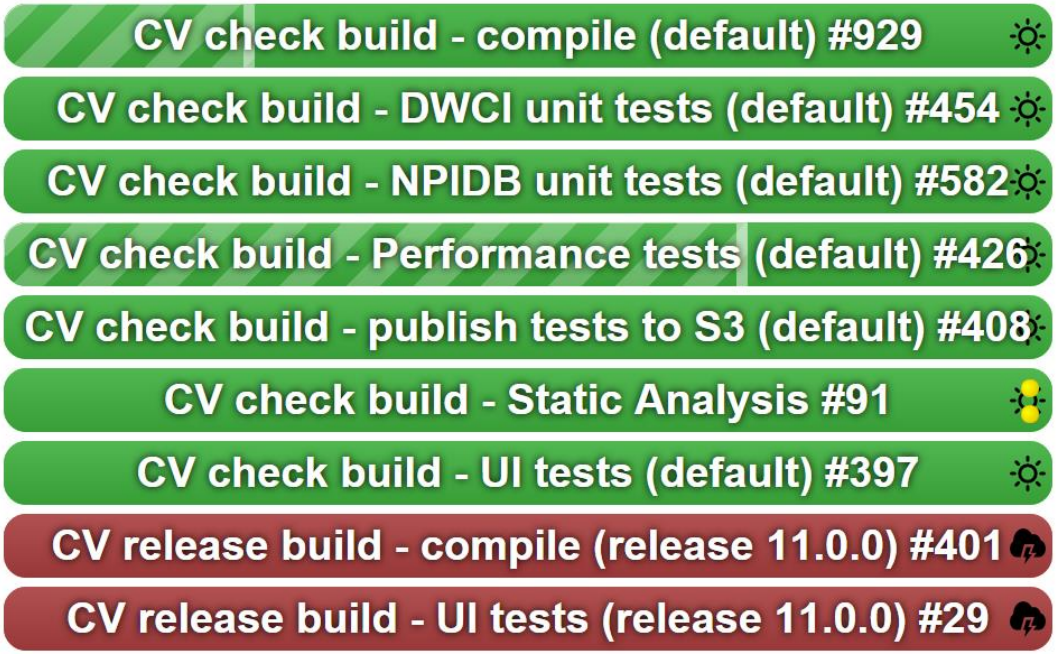
Kuva 1. Vikojen korjauksien kesto eri kehityksen vaiheissa.

Kuvassa 2 näkyvässä järjestyksessä ensimmäiseksi kehittäjä tekee muutoksen lähdekoodiin. Ennen muutoksen lisäämistä versionhallintaan, pitää kehittäjän päivittää oma työkopionsa, kääntää lähdekoodi, ajaa yksikkötestit sekä staattinen analyysi. Kun kaikki on hyväksyttävällä tasolla, voidaan muutos lisätä versionhallintaan. Mitä pienempiä muutokset ovat, sitä useammin muutoksia voidaan lisätä versionhallintaan ja sitä paremmin mahdollisilta ristiriidoilta vältytään. Jatkuvassa integraatiossa suositellaankin vähintään yhtä muutoksen lisäystä päivässä.



Kuva 2. Jatkuva integraatio

Jatkuvan integraation kaikki toimenpiteet pitää tapahtua korkeintaan yhden napin painalluksella, mielellään kuitenkin täysin automaattisesti. Tämän voi toteuttaa CI-palvelimen avulla. CI-palvelin tarkistaa yhden versionhallinnan haaran muutosvirtausta ajastetusti noin kymmenen minuutin välein. Aika on kuitenkin käyttäjän määritettävissä. Mikäli haaraan on tullut muutos, pitää lähdekoodi tarkistaa. Muutoksien tarkistus takaa sen, että mahdolliset virheet tiedotetaan kehittäjälle heti muutoksien tekemisen jälkeen, kun tieto, mitä on muutettu, on vielä muistissa. Näin lähdekoodi pysyy ehjänä ja tarvittavat kirjastot ovat lisättyinä sekä lähdekoodi on käännettävissä puhtaassa ympäristössä. Tarkistus tapahtuu kääntämällä koko lähdekoodi binaariksi käännösohjelmistolla, jota kontrolloidaan skripteillä. Skriptit suoritetaan jatkuvan integraation palvelimen avulla. Onnistuneesta käännöksestä luodaan asennuspaketti myös skriptien avulla, jota hyödynnetään automaattitesteissä. Jos automaattitestien tulokset ovat hyväksyttyjä, tiedotetaan siitä tiimille CI-palvelimen avulla esimerkiksi kuvan 3 mukaisesti.



Kuva 3. CI-palvelimen monitorointi

Viimeisimpien muutoksien myötä rakennettu asennuspaketti pitää saada helposti käyttöön vähintään testaajille. Varhainen testaus pienentää riskiä käyttöönottovaiheen löydöksille, sekä mahdollisesti pienentää korjaavien toimenpiteiden määrää. Testausta ei tule tapahtumaan pienemmille muutoksille, jos asennukseen pitää käyttää paljon aikaa. [5]

4.1.2 Testaus

Lähdekoodin testaukseen sekä käännöksestä saatavalle asennuspaketille on eri testausvaiheita. Testausvaiheita ovat yksikkötestaus, integraatiotestaus, regressiotestaus ja monia muita. Kaikkia näitä pystyy ajamaan myös automaattisesti CI-palvelimen avulla, mutta liikaa luottoa ei tämän kaltaiselle toiminnallisuudelle kannata antaa.

Yksikkötestaus ajetaan heti lähdekoodin käännöksen jälkeen. Yksikkötestaus tarkoittaa lähdekoodin pienien osien, esimerkiksi olion metodin, ajamista testiohjelmistolla, joka on kirjoitettu lähdekoodiin mukaan. Yksikkötestaus palauttaa jokaisesta eri testistä arvon tosi tai epätosi riippuen siitä, onko testi mennyt läpi. Useat yksikkötestauksen ohjelmistokomponenttikirjastot tarjoavat vielä informaatiota minkä takia testi ei ole mennyt läpi. Esimerkiksi vertailuoperaatio voi tulostaa virhetilanteessa odotetun arvon sekä oikean arvon.

Käyttöliittymätestausta kannattaa automatisoida, jos itse käyttöliittymän muutokset ovat tehtyinä. Nämä testit ajetaan ympäristössä, joka vastaa lopullista ympäristöä käyttäjän koneella. Testauksen automatisointi tapahtuu integraatiotestauksen jälkeen, milloin testaaja on varmistanut että toiminnallisuus vastaa vaatimuksia. Automatisoitua testausta voidaan pitää siis regressiotestauksena, joka tarkistaa että toteutettu toiminnallisuus ei muutu käyttäjälle näkyviltä osilta. Automaattitestaus palauttaa tuloksen, yleensä kokonaisluku josta voidaan tulkita testin tuloksen, sekä tulostaa raportin, jota tarkkaillaan CI-palvelimen avulla. [6]

4.2 Hyödyt ja haitat

Ketterät menetelmät ja jatkuva integraatio sopeutuvat mainiosti yhteen. Ketterissä menetelmissä yritetään saada tehtäviä mahdollisimman pieniksi että integraatioita olisi mahdollisimman paljon. Jatkuvan integraation prosessi varmistaa nämä pienimmätkin tuotokset, milloin järjestelmä pysyy ehjänä. Myös asennuspaketin luominen vie huomattavasti vähemmän aikaa kun se luodaan automaattisesti ja uudet testaukset voivat alkaa aikaisemmin.

Jatkuvan integraation käyttöönoton jälkeen virheet korjataan nopeammin. Virheiden tyypit ovat puhtaasti ohjelmointivirheitä eli syntaksivirheitä, tai integraation tuomia lisähaasteita. Automaattitestien avulla saadaan vielä perustoiminnot käytyä läpi, mikä antaa testaajille enemmän aikaa laajemmalle testaukselle, missä yritetään saada järjestelmää rikkovia toimintoja suoritettua.

Jatkuvan integraation käyttöönotto sekä ylläpito vaativat aluksi vähintään yhden kehittäjän työpanoksen. Virheitä syntyy CI-palvelimen operaatioista, ja virheiden hakeminen saattaa olla hankalaa ja aikaa vievää. Aikaa myöten järjestelmä kuitenkin stabiloituu, eikä prosessi varoita muista kuin lähdekoodiin liittyvistä virheistä.

5 Suorituskyvyn mittaus

5.1 Tavoitteet

Suorituskykymittauksen tavoitteena on kehittää järjestelmä joka toimii automaattisesti CI-palvelimella ja raportoi välittömästi uuden ohjelmistojulkaisun suorituskykymittauksen tulokset. Järjestelmä asentaa ohjelmiston joka sisältää viimeiset versionhallinnasta löytyvät muutokset ja testaa yleisimmiksi arvioituja toimintoja. Mittauksen tuloksista saadaan selville nykyinen tilanne, sekä miten ohjelmistoon tehdyt muutokset vaikuttavat suorituskykyyn. Tuloksia myös verrataan testattavan ohjelmiston edellisen version mittaustuloksiin.

5.2 Työkalut

Ohjelmisto asennetaan Sikulilla, joka on työpöytätoimintoja mallintava ohjelma. [7] Itse testi toteutetaan TestCompletella, joka on myös työpöytätoimintoja mallintava ohjelma ja on suunniteltu automaattisiin testeihin. [8] Tuloksien keräämiseen käytetään perfPublisher-plugin:a, joka muodostaa diagrammit mitatuista suorituskykyajoista. [9] Diagrammeista muodostetaan PhantomJS-ohjelman ja siihen tehdyn plugin:n avulla käännöskoneelle kuvatiedostoja. [10] Näihin kuvatiedostoihin muodostetaan otsikot ImageMagick-työkalun avulla, jotta näkymä olisi mahdollisimman selkeä. [11] Kuvatiedostot siirretään NAS-palvelimelle josta Agile-järjestelmä JIRA hakee kymmenen minuutin välein automaattisesti tiedostoja, jotka julkaistaan JIRA:n projektin aloitussivulla. Kaikkia näitä työkaluja ohjataan Jenkins-palvelulla batch-skriptien avulla.

5.3 Toteutus

Suorituskykymittaus suoritetaan skriptillä, joka on kirjoitettu JScript-kielellä. Skriptitiedostossa on sekä aloitus- että lopetusfunktiot ajastimelle. Lopetusfunktiossa kutsutaan kolmatta funktiota nimeltään WriteToFile, joka kirjoittaa testin nimen, ajastimen arvon sekä differenssiarvon muistiin. Differenssiarvo, joka on mitatun ajan ja ohjelmiston edellisellä versiolla mitatun tuloksen prosentuaalinen ero, on myös laskettu tässä funktiossa.

Skripti-tiedostoja ja funktioita ajetaan TestComplete projekteista. Projektit sisältävät algoritmeja, jotka mallintavat käyttäjän tekemiä toimintoja. Esimerkiksi potilaan haussa ensimmäiseksi käynnistetään CliniView, jonka jälkeen odotetaan dialogia, jossa potilasta voi hakea. Kun dialogi on ilmestynyt näkyviin, kirjoitetaan sukunimi-kenttään testin vaatima potilas ja painetaan search-painiketta. Samalla kun search-painiketta on painettu, käynnistetään ajastin kutsumalla funktiota. Tässä tilassa odotetaan, kunnes dialogin listaukseen ilmestyy haettu potilas. Kun haettu potilas ilmestyy, pysäytetään ajastin, suljetaan dialogi sekä CliniView ja siirrytään seuraavaan projektiin.

Jokainen projekti käynnistetään projekti sarjasta, joka sisältää projektin nimen, käynnistettävän tiedoston projektista sekä mahdolliset parametrit, joita projekti vaatii toimiakseen. Jotta mittaus voidaan suorittaa, tarvitaan CliniView, tietokanta sekä tietokannan sisältö testiä varten asennettuna. TestComplete ei pysty suorittamaan näitä, koska asennuksessa käytetään InstallShield-ohjelmaa ja testitietokanta asetetaan HTML-sivun kautta.

Asennuksiin käytämme Sikuli-ohjelmaa, joka käyttää kuvatunnistusta graafisessa käyttöliittymässä. Toisin sanoen Sikuli etsii tekstiä, painiketta, kuvaa tai muita objekteja siitä näkymästä, minkä käyttäjä näkee. Sikuli-projekteja on kolme. Ensimmäinen asentaa CliniView:n, toinen asettaa tietokannan käyttäjähallinnan tarvitsemat tiedot ja kolmas palauttaa testitietokannan käyttöön. Sikuli-projektit myös täyttävät automaattisesti lokitiedostoon tehdyt toiminnat, esimerkiksi ”painikkeen painaminen suoritettu”.

Kaikki edellä mainittu toimii virtuaaliympäristössä. Virtuaaliympäristöä käytetään sen alustamisen helppouden takia. Virtuaalikoneelle on asennettu TestExecute, Sikuli, VLC player sekä testitietokannan aktivointiin vaaditut tiedostot. [12] Ennen testien käynnistämistä, virtuaalikone palautetaan snapshotiin, missä on tarvittavat tiedostot ja työkalut asennettuina.

Virtuaalikonetta ja sen sisällä tapahtuvaa toiminnallisuutta ohjataan CI-palvelimen Jenkins -palvelun kautta. Jokainen toiminto jonka Jenkins käskee suoritettavaksi, palauttaa arvon, joka kertoo, onko tapahtunut virheitä vai onko toiminta edennyt, kuten on suunniteltu. Jos toiminnassa on tapahtunut virhe, Jenkins tunnistaa tämän ja lopettaa testien suorituksen, kopioi virheen etsintää helpottavat tiedostot artifaktoihin, eli säilytettäviin tiedostoihin, ja siirtää projektin hälytystilaan. Hälytystilassa oleva projekti lähettää

automaattisesti sähköpostin muutoksia tehneille kehittäjille, että korjaavat toimenpiteet alkaisivat mahdollisimman pian.

5.4 Mittauksien kohteet

Mitattavia kohteita ovat CliniView:n toiminnallisuudet, joita loppukäyttäjä suorittaa paljon. Näihin kuuluvat potilashaku, kuvakansion avaus, kuvan avaus sekä kuvien tallennus.

Potilashaussa mitataan kahta erilaista toimenpidettä. Ensimmäisessä testissä hakukriteereitä lisätään niin, että potilaslistaan palautuu vain yksi potilas. Toisessa kriteereitä ei käytetä lainkaan, jolloin potilaslistaan palautuu 100 potilasta. Enemmän käytössä oleva haku kriteerien kanssa on tärkeämpi arvo, mutta useamman potilaan haku voi tuoda lisäinformaatiota suorituskykyä parantaessa.

Kuvakansion avaus tehdään kahdelle eri potilaalle. Kuvakansio avautuu automaattisesti, kun potilashaku-lomakkeelta valitaan potilas ja painetaan Open-painiketta. Toiminnallisuus toistetaan aina, kun potilaalle otetaan uusia kuvia tai otettuja kuvia käytetään diagnosointiin.

Jotta kuvasta voitaisiin tehdä päätelmiä, on kuva avattava. Myös tätä toiminnallisuutta mitataan. Aika mitataan kuvan avauskäskystä siihen, että kuva ilmestyy suurempana näytölle. Myös kymmenen kuvan avaus mitataan samasta aloituskohdasta siihen, että kymmenen kuvaikkunaa on avautunut.

Kuvien tallennuksessa mitataan kolmea erilaista tapaa. Ensimmäisenä on yksittäisen kuvan tallennus. Kuvan tallennus tässä tapauksessa on jo tallennetun kuvan päivitys tietokantaan. Toisessa mittauksessa tallennetaan kymmenen kuvan muutokset tietokantaan eli sama testi kuin ensimmäinen, mutta suuremmalla volyymilla. Kolmas tallennusmittaus tapahtuu 18 kuvan sarjalle, jotka lisätään ensimmäistä kertaa tietokantaan.

6 Testauksen kattavuus

Yksikkötestausta käytetään ohjelmiston kehitys- ja ylläpitovaiheessa. Tämä nopeuttaa kehitystä sekä parantaa ohjelmiston laatua. Testejä tehdään ohjelmiston pienimmille yksiköille, kuten olion tarjoamille metodeille. Testejä tehdessä saattaa kuitenkin jäädä huomioimatta joitakin metodin alueita, joita kattavuusmittauksen avulla pyritään tuomaan esille. Ylläpitovaiheessa muutoksia tehdessä saattaa vahingoittaa useampaa toiminto pienellä muutoksella, mutta yksikkötesteissä nämä jää helpoiten kiinni. Jos yksikkötestejä on useassa eri tasossa, tämä myös helpottaa huomioimaan missä virheellinen toiminto tapahtuu.

Testauksen kattavuus on mittausarvo, jota käytetään kuvaamaan kuinka paljon lähdekoodia testiohjelma käy läpi. Kattavuutta mitataan työkalulla, joka instrumentoi suoritettavan ohjelman joko lähdekoodia kääntäessä tai jo käännetyn koodin binääritiedostojen avulla. Instrumentointi auttaa työkalua lukemaan millä lähdekoodin alueilla on käyty. Yksikkötestauksen ajettava versio käynnistetään työkalun kautta ja suoritetaan loppuun. Työkalu kerää tiedon kattavuudesta ja tulostaa raportin, missä on yhteenveto kattavuudesta sekä yksityiskohtaisempaa tietoa tiedostoista rivitasolle asti.

6.1 Tavoitteet

Kattavuusmittauksen tavoitteena on suorittaa mittauksen automaattisesti CI-palvelimella. Mittauksien tulokset julkaistaan, jos suoritus toimii hyväksytysti, muuten kerätään virheiden etsimiseen tarkoitettu tieto. Hyväksytyistä suorituksista kerätään kattavuusluku ja lisätään viivakaavioon.

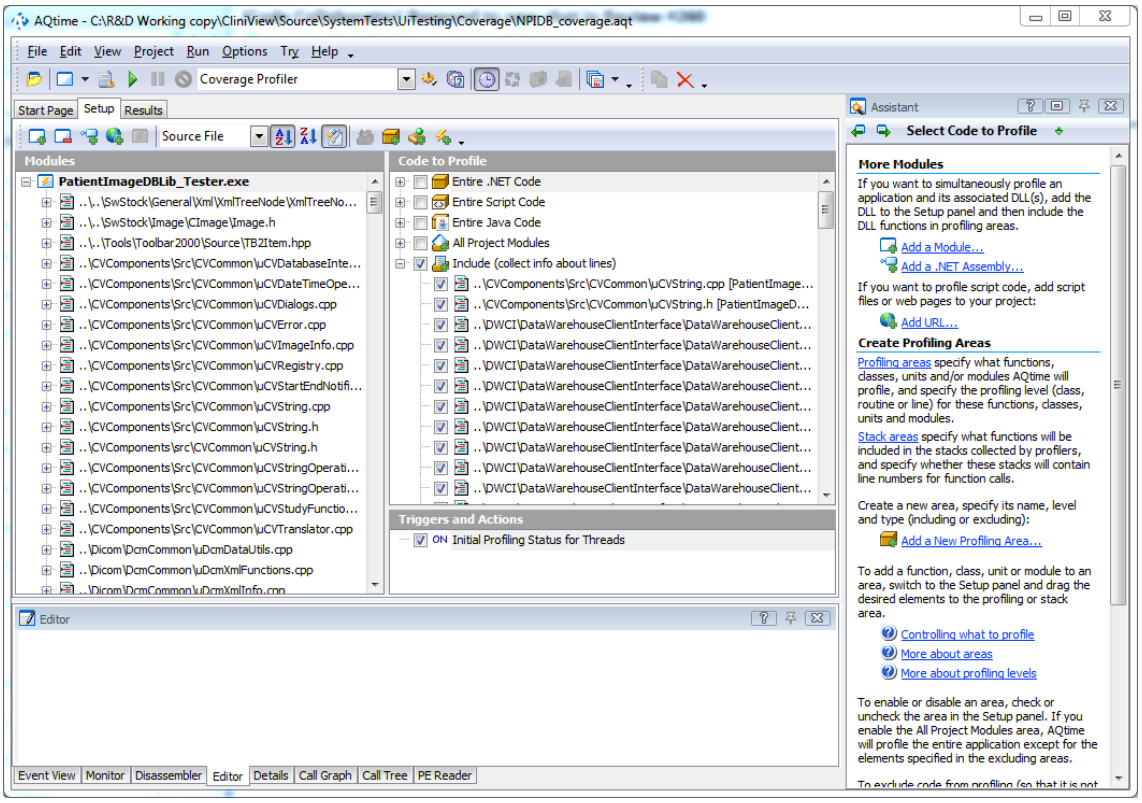
6.2 Työkalut

AQTime tarjoaa mahdollisuuden kattavuuden profilointiin joko oman käyttöliittymän kautta, sekä integroituu myös Embarcadero XE5 kääntäjään. [13] AQTime:n ohjauksen pystyy suorittamaan COM-rajapinnan kautta, joka mahdollistaa käytön automaattisesti käännöskoneella.

6.3 Toteutus

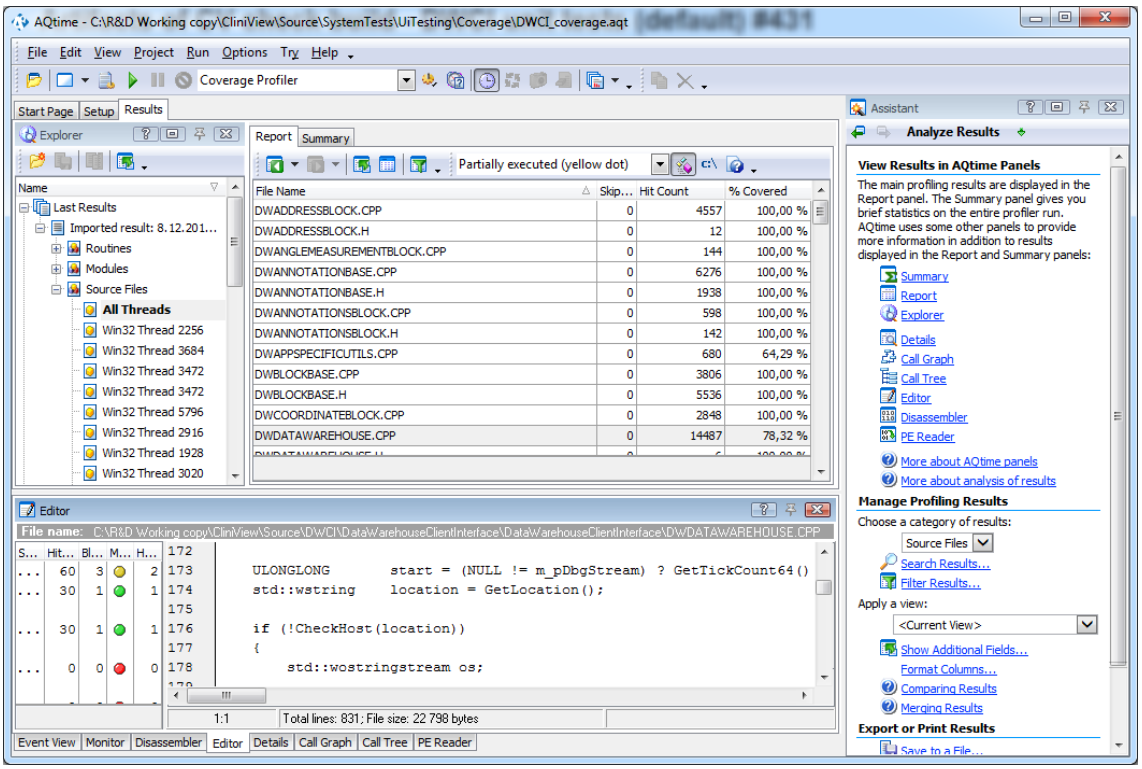
Lähdekoodia testataan yksikkötestauksella kahdella eri tasolla, minkä vuoksi AQTime:lla luodaan kaksi projektia kattavuusmittauksen suoritukseen.

Ensimmäiseksi projekti tarvitsee moduulin profilointia varten. Moduuliksi lisätään käännetty yksikkötestauksen suorittavan projektin ajettavan tiedosto (.exe). AQTime tunnistaa automaattisesti moduulista lähdekoodin luokat ja luokkien metodit kuvan 4 mukaisesti. Seuraavaksi luodaan ryhmä, johon lisätään luokat mistä kattavuutta halutaan mitata.



Kuva 4. AQTime asetukset. Vasemmalla sijaitsevat modulit ja keskellä ryhmät.

Ryhmän asetuksissa on mahdollisuus asettaa profilointi keräämään tietoa joko metodi- tai rivitasolla, joista valitsemme rivitason. Kun halutut luokat on lisätty, ajetaan profilointi, joka suorittaa exe:n tietoa keräten. Suorituksen jälkeen AQTime esittää automaattisesti tulokset kuvan 5 mukaisesti. Tulokset jaetaan rutiineihin, moduuleihin sekä lähdekoodi tiedostoihin ja näiden alapuolelta löytyy vielä käytetyt säikeet.



Kuva 5. AQtime tulokset. Vasemmalla tuloksien näyttämisvalinta sekä säikeet

Tässä vaiheessa olemme kiinnostuneita tiedostojen kattavuudesta kaikissa säikeissä, jotka aktivoimme nähdäksemme tuloksista mitä tiedostoja on käytetty.

Tuloksien tiedostoista löytyy kääntäjän käyttämiä peruskirjastoja, joihin emme voi vaikuttaa. Nämä tiedostot poistetaan tuloksista lisäämällä tiedostojen nimet estolistaan. Testiajoja sekä estolistan päivitystä jatketaan niin pitkään, että tarvittavat tiedostot löytyvät tuloksista eikä peruskirjastoja ole näkyvissä.

CI-palvelin suorittaa ensimmäisenä koko järjestelmän rakennustoimenpiteet. Näissä toimenpiteissä rakennetaan myös yksikkötestausprojektit sekä Coverage-projekti jota käytetään AQTime:n ohjaukseen. Coverage sisältää toiminnallisuuden, joka ensin puhdistaa mittauksen aikana käytettävät kansiot ja kopioidaan tarvittavat tiedostot mittausta varten. Yksikkötestauksen ajoa varten on muutettava sekä ajettavan että debug informaatio tiedoston muokkausaikaleimat samoiksi. Muuten AQTime antaa varoituksen välttääkseen eri käännöksien tiedostojen käyttöä samassa mittauksessa. Varoitus annetaan jos ajettavan ja debug informaatio tiedostojen aikaleimojen ero ylittää 30 sekuntia. Tarkoituksena on ajaa järjestelmää CI-palvelimella, joten haluamme välttää ylimää-

räisiä virheitä. Aikaleimojen muokkauksen jälkeen siirrytään komentamaan AQTime profiloijaa.

AQTimea on mahdollista käyttää automaattisesti COM-rajapinnan kautta kuvan 6 mukaisesti. Ensimmäiseksi yritetään luoda yhteyttä olemassa olevaan instanssiin Marshall-luokan kautta. Jos olemassa olevaa instanssia ei löydy, luodaan uusi Activator rajapinnan kautta käyttäen AQTimen identifiointi numeroa, mikä on tallennettu rekistereihin. Jos instanssia ei vieläkään ole, lopetetaan ohjelma ja tulostetaan tekstitiedostoon virheen syy.

```
// Obtains access to AQtime
try
{
    AQtimeObject = Marshal.GetActiveObject(AQtimeProgID);
}
catch
{
    try
    {
        AQtimeObject = Activator.CreateInstance(Type.GetTypeFromProgID(AQtimeProgID));
    }
    catch
    {
        exitvalue = 2;
    }
}
```

Kuva 6. Coverage.exe AQTime instanssin aktivointi

Mikäli instanssi saadaan käyttöön, avataan AQTimeManager:lla projekti, joka sisältää tiedot mistä mitataan ja mitä ei mitata. AQTimeManager:n IntegrationManager:lla syötetään ohjelmalle profiloinnin tyypiksi Coverage profiler eli kattavuus kuvan 7 mukaisesti.

```
// Loads the project
if (!IntegrationManager.OpenProject(AQTimeProject))
{
    exitvalue = 3;
}
else
{
    // Selects the desired profiler
    if (!IntegrationManager.SelectProfiler("Coverage Profiler"))
    {
        exitvalue = 4;
    }
```

Kuva 7. AQTime projektin avaus ja profiloijan valinta

IntegrationManager sisältää myös suoritustyypin RunMode, joka asetetaan normaaliksi että mahdolliset virheet eivät jää piiloon kuvan 8 mukaisella tavalla. AQTimelle annetaan yksikkötestauksen suoritustiedoston polku että huomioon otetaan myös uusimmat muutokset lähdekoodissa.

```
// Obtains aqTimeIntegrationRunMode object for the "Normal" profiling mode
AQtime.IaqTimeIntegrationRunMode RunMode = IntegrationManager.GetRunMode(4);

aParamValue = null;
// Gets the "Work Directory" parameter
aParamValue = RunMode.GetParameterValue(2);

// If the parameter is empty assigns a value
if (aParamValue == null) RunMode.SetParameterValue(2, workingFolder);

// Selects "Normal" profiling mode
IntegrationManager.SelectRunMode("Normal");

// Adds new module to AQtime project
IntegrationManager.AddModule(exePath);
```

Kuva 8. AQTime asetukset profilointia varten

Kun asetukset on asetettu, käynnistetään profilointi start-komennolla. Start-komento ottaa vastaan kaksi string tyypin parametria, joista ensimmäiseen sijoitetaan polku yhteenveto tuloksia varten kuvan 9 mukaisesti. Toiseen parametriin voisi sijoittaa tiedot myös riveistä, joissa on käyty, mutta tämä jätetään tyhjäksi koska käytössä on toinen raportointitapa.

```
// Starts profiling and saves results to xml files
IntegrationManager.Start(summaryFolder + "\\SummaryResults.xml", "");

// Waits until profiling is over
while (IntegrationManager.ProfilingStarted)
{
    Application.DoEvents();
}
```

Kuva 9. AQTime käynnistys

Kun profilointi on valmis, viedään AQTime:n oma raportti samaan kansioon minne myös yhteenveto tiedosto sekä poistetaan suoritustiedosto AQTime:sta kuvan 10 esittämällä tavalla.

```
// Exports .aqr result file
IntegrationManager.Results.ExportResult(summaryFolder + "\\Coverage.aqr",
                    IntegrationManager.Results.LastResultName[0].ToString());

// Deletes result file from AQTime project
IntegrationManager.Results.DeleteResult(IntegrationManager.Results.LastResultName[0].ToString());

// Removes previously added module
IntegrationManager.RemoveModule(exePath);
exitvalue = 0;
```

Kuva 10. AQTime tuloksien vienti sekä projektin puhdistus

Profiloinnin jälkeen suljetaan AQTime sekä vapautetaan alussa käyttöön otettu instanssi kuvan 11 mukaisesti, jotta muut ohjelmat voivat jatkaa suoritusta.

```
// Closes AQtime
AQtimeManager.Quit();

// Releases COM objects
Marshal.ReleaseComObject(IntegrationManager);
Marshal.ReleaseComObject(AQtimeManager);
Marshal.ReleaseComObject(AQtimeObject);
```

Kuva 11. AQTime COM-instanssin vapautus

Koko coverage projektin suoritettava koodi on ympäröity try-catch ohjelma-lohkoilla, jotka katkaisevat toiminnan virheen sattuessa sekä kirjoittavat tiedostoon kuvauksen virheestä, joita on esitetty kuvassa 12. Esimerkiksi jos AQTime projektin lisääminen IntegrationManager-luokalle ei onnistu, tulostetaan tiedostoon teksti: Cannot open the specified project. Virhetilanteissa nämä tiedot helpottavat vianetsintää.

```
switch (result)
{
    case 0:
        errorMessage = "0";
        break;
    case 1:
        errorMessage = "Wrong parameters";
        break;
    case 2:
        errorMessage = "Cannot obtain access to AQTime";
        break;
    case 3:
        errorMessage = "Cannot open specific project";
        break;
    case 4:
        errorMessage = "Specific profiler was not found";
        break;
    default:
        errorMessage = "Unknown error";
        break;
}
```

Kuva 12. AQTime virhetilanteiden tulosteet

Virhetilanteet tulostetaan tekstitiedostoon CI-palvelimen oikeustason takia. AQTime vaatii järjestelmävalvojan oikeudet suoritukseen, jotka lisäämme Tehtävien ajoitusohjelman avulla. Tehtävien ajoituksen kirjastoon pitää luoda kaksi tehtävää, molemmille yksikkötestauksille omansa. Tehtävän yleiset-välilehdellä asetetaan nimi tehtävälle sekä asetetaan tehtävä käyttämään toimintoa laajimmilla mahdollisilla käyttöoikeuksilla. Toiminnot välilehdelle lisäämme käynnistä ohjelma toiminnon, johon ohjelman poluksi asetetaan Coverage.exe:n käynnistävän batch-tiedoston osoitteen. Ajoitettu tehtävä käynnistää batch-tiedoston, koska coverage.exe:lle välitetään käynnistymisparametreja, ja näitä haetaan Windows:n ympäristömuuttujista. Ajoitettu tehtävä käynnistetään Jenkins-palvelusta, joka asettaa ensin tarvittavat ympäristömuuttujat. Ympäristömuuttujiin lisätään tuloskansion sijainti, GoogleTest-framework:n xml-tulostukseen vaaditut attribuutit sekä yksikkötestauksen suoritustiedoston sijainti.

CI-palvelin hakee kattavuusmittauksen tekemät tiedostot ja tallentaa ne omiin julkaistaviin tiedostoihin. Yhteenveto-tiedostosta haetaan myös kokonaiskattavuus-arvo, joka sijoitetaan Jenkinsissä sijaitsevaan PlotPlugin-työkalun tekemään viivakaavioon. Gtest-xml käsitellään myös Jenkins-työkalulla, missä tarkistetaan suoritettujen testien tila. Yksikkötestaus järjestelmässä on useita testejä, joiden tila voi suorituksen jälkeen olla joko hyväksytty tai hylätty. Jos yksikin testi on hylätty, testaus hylätään. Työkalu muo-

dostaa automaattisesti kaavion, josta näkee suoritettujen, hyväksyttyjen ja hylättyjen testien määrän.

Säilytettävistä tiedostoista löytyy myös CoverageSummary.aqr-tiedosto, joka on AQTime:sta tuloskansioon viety raportti. Raportin pystyy tallentamaan Jenkins-palvelusta omalle koneelle ja avaamaan AQTimella Embargadero XE5 kääntäjän kautta. Tuloksista selviää millä riveillä on käyty. Kääntäjässä on ensin avattava AQtime:n näkymä, minne tiedoston saa tuotua. Kun tiedosto on tuotu, näkymässä näkyvät samat routines, modules ja source joissa jokaisessa säielistaus. Nämä saa avattua toiseen ikkunaan, mistä näkyy luokan ja rivin käyntimäärät. Valitsemalla rivin, kääntäjä avaa lähdekooditiedoston ja asettaa rivin näkyviin. Näin kehittäjä voi parantaa testausta ja nostaa testauksen kattavuutta.

7 Staattinen analyysi

Ohjelmistokehityksen monimutkaiset algoritmit altistavat erilaisille koodivirheille. Virheiden etsiminen saattaa kestää pitkään ja tuo sitä enemmän kustannuksia mitä myöhäisemmässä vaiheessa ne ilmenevät. Nämä virheet pyritään poistamaan jo koodia kirjoittaessa tai ainakin mahdollisimman aikaisessa vaiheessa erilaisien työkalujen avulla. Valmiita työkaluja on tarjolla varsin hyvin muun muassa Parasoft C/C++test, AQTime, CppCheck ja niin edelleen. Työkalun prosessia kutsutaan nimellä staattinen analyysi. Staattisen analyysin merkitys tulee erityisen olennaiseksi turvakriittisissä sovelluksissa, joissa koodinlaadulle kohdistuu ulkopuolelta usein tiukkoja vaatimuksia.

Staattisella analyysilla tarkoitetaan virheiden etsimistä lähdekoodista ilman ohjelman suorittamista. Jos ohjelma ajetaan analyysin aikana, sitä kutsutaan dynaamiseksi analyysiksi. Työkalut analysoivat kaikkia loogisia funktioiden polkuja ja pystyvät huomattavasti parempaan kattavuuteen kuin perinteiset testaustyökalut tai kääntäjä itse. Työkalut etsivät koodin heikoimpia kohtia, muistivuotoja, datan virheellistä muokkausta, puskurimuistin ylityksiä, koodin kompleksisuutta ja paljon muuta. Yleensä analyysin suoritus ei kestä pitkään verrattuna siihen, kuinka paljon erilaisia löydöksiä tämä voi tuottaa.

Työkalujen analyysin suoritus eroavat toisistaan jonkin verran. Toiset tarkistavat yksittäiset rivit ja etsivät mahdollisia virheitä, kun taas toiset sisällyttävät koko lähdekoodin analyysiin.

Staattisen analyysin työkaluilla on myös ongelmakohtia. Työkalut voivat tulostaa myös virheellistä tietoa. Näitä kutsutaan vääriksi positiivisiksi ja vääriksi negatiivisiksi. Väärä positiivinen tarkoittaa virheen raportoimista, kun virhettä ei oikeasti ole olemassa. Väärä negatiivinen tarkoittaa virheen raportoinnin puuttumista. Työkalujen on haettava oikeaa taitekohtaa tarkkuuden ja suorituskyvyn välillä. Mikäli työkalu on erittäin tarkka, raportissa on erittäin paljon vääriä positiivisia löydöksiä ja analysointiaika kestää pitkään. Toinen ääripää eli suorituskykyinen analysointi jättää virheitä huomioimatta. Näitä asioita pystyy myös hallitsemaan työkalun asetuksissa. [14]

7.1 Tavoitteet

Tavoitteena on etsiä staattisen analyysin työkalu, jota pystyy ajamaan CI-palvelimen kautta sekä kehittäjän omassa kehitysympäristössä.

CI-palvelimen kautta työkalun käyttö tapahtuu batch-skriptien avulla. Käynnistys tapahtuu toisen skriptin kautta, mikä lisätään versionhallintaan muutoksien jäljitettävyyden takia. CI-palvelimella staattisen analyysin kattavuus sisältää koko lähdekoodin, kuitenkin poislukien kolmannen osapuolen työkalut.

Kehitysympäristössä ei käsitellä koko lähdekoodia, joten työkalulla tarkistetaan vain muutamia lähdekooditiedostoja. Mitä vähemmän työtä analyysin suorittaminen vaatii, sitä enemmän sitä käytetään. Parhain mahdollinen käynnistystapa on suoraan kääntäjän kautta.

Tuloksien raportointi tapahtuu CI-palvelimella. Tuloksista näkee muutokset huomautusmäärissä, sekä näiden historiatiedot. Raportista näkee myös huomautustyypin sekä missä virhe tapahtuu.

7.2 Työkalut

AQTime Pro sisältää static analysis-profiloinnin, mutta ei vastaa odotuksia. Analyysin tuloksista näkyy tarkasteltavan luokan lukumäärät kutsuvista sekä kutsuttavista luokista, kutsumäärän sekä suoritusjonojen lukumäärät. Tietoa halutaan mahdollisista koodivirheistä, joten AQTime ei sovellu käyttötarkoitukseen.

Cppcheck on avoimen lähdekoodin työkalu joka ei tarkista lähdekoodin syntaksivirheitä. Työkalu keskittyy havainnoimaan virheitä, mitä kääntäjät eivät huomioi. Cppcheck:ä pystyy ajamaan sekä graafisen käyttöliittymän kautta että komentorivillä tiettyjen komentojen avulla. Komentojen avulla raportin voi viedä xml-tiedostoon, jonka saa jälleen avattua Cppcheck:n käyttöliittymän kautta, jolloin tuloksia on huomattavasti helpompi tarkastella. Cppcheck tarkistaa jokaisen luokan koodin, poikkeushallinnan, muistivuodot, käyttämättömät luokat ja muuttujat, vääränlaisen STL-kirjaston käytön sekä tarkastaa raja-arvojen ylitykset. [15]

Parasoft C/C++test:n lähtöhinta alkaa 2500 dollarista, joten valintana ensimmäiseksi työkaluksi päätimme jättää sen pois vertailusta. Työkaluksi valitsimme Cppcheck:n.

7.3 Cppcheck konfiguraatio

Staattisen analyysin profilointialueeksi lisätään koko lähdekoodi, josta poistetaan tämän jälkeen kolmannen osapuolen kirjastot yksitellen. Tiedostojen sekä kansioiden analyysista poistaminen tapahtuu lisäämällä komentoriville ”–i”-attribuutin ja sen arvoksi poistettavaksi halutun tiedoston polun. Cppcheck ei ymmärrä lähdekoodin kaikkia symboleita, jotka voi poistaa käyttämällä ”-U”-attribuuttia, jonka arvoksi lisätään symbolin nimi.

Staattisen analyysin suoritusaikaa pyritään pienentämään säikeistämällä tiedostojen analysoinnit neljään eri säikeeseen, joka on suositeltu määrä.

Tulokset viedään XML-tiedostoon komennolla ”--xml --xml-version=2 2<cppcheck.xml”. XML-tulostus otetaan käyttöön attribuutilla ”--xml”. Käyttöönoton jälkeen valitaan formaatti XML-tiedoston sisällölle, minkä suositus on versio 2. Versio-informaation jälkeen siirretään tieto XML-tiedostoon.

Tuloksien määrää pienennetään aluksi tarkoituksella. Jos löytöjen määrä on liian iso, syntyy epäselvyyttä kehittäjille, mitä pitäisi ensisijaisesti korjata. Aluksi aktivoidaan vain vakavat virheet, varoitukset, suorituskykyhuomiot sekä siirrettävyys. Mahdollisuus olisi vielä ottaa mukaan tyylivirheet sekä analyysin informaatiot, joiden käyttöönotto siirretään myöhemmäksi.

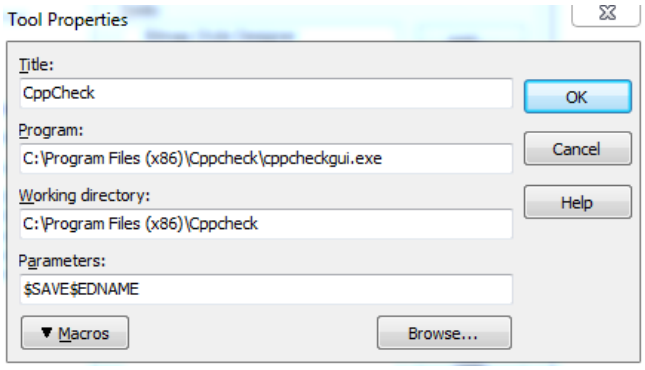
Xml-tiedoston pystyy avaamaan Cppcheck ohjelmalla, tai tuomaan ohjelmaan ohjelman omien valikoiden kautta. Näin käyttäjä saa tietää mitä virheitä löytyy ja missä ne toistuvat. Myös CI-palvelin käyttää xml-tiedostoa tuloksien näyttämiseen. CI-palvelimella on Cppcheck-lisäosa, joka tuo samat tiedot virheistä kuin Cppcheck:n graafinen käyttöliittymä, mutta näyttää myös lähdekoodin kun virheen valitsee.

7.4 Toteutus

Cppcheck asennetaan ensin haluttuun hakemistoon. Asennuksen jälkeen on lisättävä suoritettavan tiedoston kansion polku ympäristömuuttujiin, jotta analysoinnin voi suorittaa komentorivillä missä sijainnissa tahansa.

Cppcheck:n saa analysoimaan tiedoston, joka on kääntäjän editorissa näkyvillä. Kääntäjään pitää lisätä uusi työkalu toimintoa varten, joka asetukset sisältää työkalun nimen, ohjelman suoritettavan tiedoston polun, työkansion sekä ohjelmalle välitettävät parametrit.

Parametreihin lisätään ensimmäiseksi ”$SAVE”-komento kuvan 13 mukaisesti, joka pakottaa tallentamaan ensin editorissa päällimmäisenä sijaitsevan tiedoston, mikäli tiedosto sisältää muutoksia joita ei ole vielä tallennettu. Tallennus tehdään sen takia, että Cppcheck ei analysoi haihtuvassa muistissa olevaa tietoa. Toiseksi parametriksi lisätään $EDNAME, joka välittää editorissa päällimmäisenä olevan tiedoston polun Cppcheck:lle.



Kuva 13. Työkalun asetuksien lisääminen kääntäjään.

Cppcheck:n asetuksiin lisätään tiedoston avaus, kun virhettä kaksoisnapsauttaa. Nimeksi asetetaan kääntäjän nimi, mutta suoritettavaksi tiedostoksi asetetaan explo-

rer.exe:n polku. Jos suoritettavaksi tiedostoksi valittaisiin kääntäjä, avautuisi uusi kääntäjä automaattisesti. Jos explorer.exe:ä käytetään, tiedosto, jossa virhe sijaitsee, avautuu automaattisesti oletusohjelman, eli kääntäjän, olemassa olevassa instanssissa, tai uudessa instanssissa mikäli kääntäjä ei ole jo auki.

Komentorivin kautta Cppcheck:ä käytettäessä on ensin mentävä polkuun, minne tulostetaan raporttitiedosto tai vaihtoehtoisesti käynnistetään valmiiksi luotu Windows batch tiedosto, mihin on määritetty koko lähdekoodin tarkistus valmiiksi. Kun analysointi on valmis, muodostuu raportti xml-tiedostoon jota pystyy lukemaan suoraan, tai raportin voi avata Cppcheck-ohjelmassa.

CI-palvelin ajaa staattisen analyysin jos lähdekoodin käännös on mennyt läpi hyväksytysti. Staattisen analyysin kohde on koko lähdekoodi lukuun ottamatta kolmannen osapuolen kirjastoja, mihin ei voi vaikuttaa. CI-palvelimen projekti sisältää työkalun, joka analysoi tulokset sijoittamalla ne virhetyypin mukaan listaan. Listassa on myös näkyvillä muutos edelliseen analysointiin verrattuna.

Tarkempia tietoja virheistä saa avaamalla tulokset kuvassa 14 näkyvällä linkillä jossa lukee ”Cppcheck Results”. Tämä avaa listan virheistä tarkemmilla kuvauksilla. Ensimmäisessä sarakkeessa on virheen tila edelliseen analysointiin verrattuna. Tila voi olla uusi, muuttumaton tai korjattu virhe.

Cppcheck Results

Severity	Count	Delta
Error	40	
Warning	172	
Style	5	
Performance	131	
Portability	9	-1
Information	0	
No category	0	
Total	357	-1

Kuva 14. Staattisen analyysin yhteenveto CI-palvelimella

Toisessa sarakkeessa kerrotaan tiedoston polku, joka toimii linkkinä lähdekoodiin. CI-palvelin avaa lähdekooditiedoston selaimeen ja korostaa rivin mistä virhe löytyy. Vie-

mällä hiiren osoittimen korostetun rivin päälle saa tarkempaa tietoa minkälainen virhe on kyseessä sekä tietoa, miten virheen voi korjata.

Kolmas sarake sisältää rivinumeron missä virhe on sekä linkin korostetulle riville lähdekooditiedostoon.

Neljännellä sarakkeella on tieto virheen vakavuudesta, jotka näkyvät kuvan 13 listassa Severity-sarakkeella.

Lyhyt versio virheen tyypistä on kerrottu viidennellä sarakkeella. Virheen tyyppejä on useita, esimerkiksi muistin varaukseen tai vapautukseen liittyvä virhe, käyttämätön muuttuja, muistivuoto ja monia muita.

Virhe voi olla myös väärä positiivinen. Näistä varoitetaan kuudennella sarakkeella arvolla tosi tai epätosi, jos Cppcheck analyysi ei pysty syystä tai toisesta tekemään tarkkaa arviointia.

Seitsemäs ja viimeinen sarake sisältää viestin virheestä. Viesti sisältää tiedon, mikä virhe on kyseessä, kuten viides sarake, mutta helpommin luettavan version. Esimerkiksi viidennen rivin tieto voi olla ”uninitvar”. Viimeisessä sarakkeessa virheestä kerrotaan, että muuttujaa ei ole alustettu sekä muuttujan nimi.

Työkalun tarjoaman listan pystyy järjestämään jokaisen sarakkeen mukaan aakkosjärjestykseen, joka helpottaa korjaamisen prioriteettia. Uusia virheitä ei hyväksytä yhtään. Jos uusi virhe havaitaan, analyysi hylätään ja muutoksia tehneille kehittäjille tiedotetaan sähköpostin avulla hylätystä ajosta.

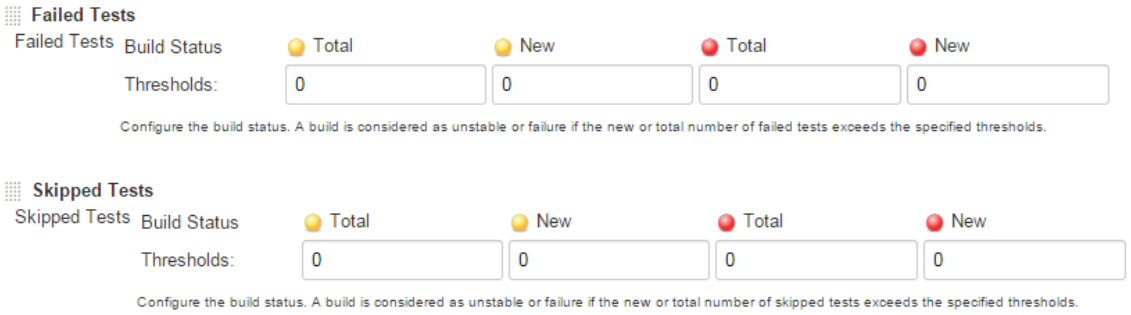
8 Liittäminen jatkuvaan integraatioon

8.1 Riskit ja niiden hallinta

Jatkuvassa integraatiossa tapahtuvien virheiden huomioimatta jättäminen on pahin riski järjestelmässä. CI-palvelu tarjoaa virheiden hallinnan hyvin pitkälle, mutta joitain asioita on jätetty palvelun käyttäjälle.

CI-palvelun projektiin sisällytetään lisäosia, joissa on sisäänrakennettu virheenhallinta-järjestelmä. Osassa lisäosista on perinteinen ”toimii tai ei toimi” -hallinta. Esimerkiksi versionhallintalisäosa, joka päivittää työkopioon viimeisimmät muutokset, hylkää projektin suorituksen jos yhteyttä palvelimeen ei pystytä muodostamaan.

Toisissa lisäosissa on mahdollista määritellä, minkälaisia virheitä sallitaan. Yksikkötestauksessa pystytään määrittämään kuinka monta testiä voidaan hylätä, ennen kuin projektin käännös asetetaan varoitus- tai hylkäystilaan kuvan 15 tavalla.



Kuva 15. Yksikkötestauksen rajamääritys

Ensin määritetään, kuinka monta testiä voi epäonnistua yhteensä ja kuinka monta uutta epäonnistunutta testiä saa olla, ennen kuin käännöksen tila muuttuu. Käännöksen tilat ovat ”onnistunut”, ”epävakaa” ja ”epäonnistunut”.

Osaan projektien toiminnoista pitää tehdä itse virheenhallinta. Näitä pitää tehdä silloin, kun toiminnallisuus suoritetaan skriptien avulla. Mikäli toiminnallisuus saadaan suoritettua vain yhdellä skriptillä, on mahdollisuus tehdä vain paluuarvot skripti-tiedostoon, ja tarkkailla tätä CI-palvelun projektissa. Mikäli skriptejä on useampia, kuten tilanteissa mitkä vaativat peruskäyttäjää korkeampia oikeuksia, pitävät paluuarvot kierrättää paikallisen tallennuksen avulla toiseen skriptiin, joka tarkkailee tiedoston ilmestymistä missä paluuarvo on.

Pitkissä skripteissä on hyvä käyttää tulosteita mahdollisimman paljon. Virheen sattuessa virheen etsintä on erittäin haastavaa, jos tietoa ei löydy missä virhe on sattunut. Yleisin virheilmoitus on syntaksivirhe, joka ei muuta informaatiota kerro. Jos virheen edellä on tuloste mitä toteutetaan seuraavaksi, päästään helpoiten jäljille mitä arvoja lähteä selvittämään.

8.2 Testaus

CI-palvelun projektien testaus on suoritettu TDD (Test Driven Development) viitekehyksen ajatusmaailmaa mukaillen. Pääosa projekteissa suorittavia toiminnallisuuksia kontrolloidaan skriptien avulla, jotta nämä saadaan lisättyä versionhallintaan ja jäljitettävyys säilyy paremmin. Ensin tehdään projekti, joka kutsuu skriptejä, ja tarkistetaan että se epäonnistuu koska tiedostoja ei ole vielä luotu. Tämän jälkeen kehitetään osa kerrallaan projektin suoritusta eteenpäin niin kauan, että suoritus onnistuu. Tämä takaa virheenhallinnan on toimivuuden.

Testattava järjestelmä muuttuu jatkuvasti ja projektin ollessa alkuvaiheessa, virheitä syntyy jatkuvasti, mutta järjestelmä stabiloituu nopeasti. Ylläpito ja jatkokehitys on alussa haastava ja raskas toimenpide.

9 Yhteenveto

Insinöörityön tavoitteena oli parantaa jatkuvan integraation laadun varmistusta suorituskykymittauksella, yksikkötestauksen kattavuusmittauksella sekä staattisella analyysilla. Tämän lisäksi kirjalliseen osuuteen oli tarkoitus tutkia ketterien menetelmien sekä jatkuvan integraation teoriaa.

Vaadittavat laadunparannustekniikat on toteutettu ja tarvittavat työkalut on arvioitu, valittu ja hankittu jatkuvan integraation toteutukseen aikataulussa. Tuotteen suorituskykyä on jo parannettu tuloksien näkyvyyden ansiosta.

Ongelmia ilmeni yksikkötestauksen kattavuusmittauksen toteutuksessa monipuolisen ympäristön takia sekä työkalujen automatisoinnin yhteydessä, mutta aikataulua saatiin kirittyä staattisen analyysin toteutuksessa runsaasti. Kattavuusmittauksessa yritin muutamaa eri työkalua, joista AQTime onnistui instrumentoimaan lähdekoodin binaarit. Tämän työkalun automatisoimista yritin komentorivin avulla, mutta komentoja ei riittänyt raportoimiseen. Suoritus onnistui COM-rajapintaa hyväksi käyttäen omalla C#-projektilla. AQTime:n varoitukset vielä estivät mittauksen tekemistä, mitkä sain korjattua projektin avulla. Staattisen analyysin työkalun, Cppcheck:n, käyttöönotto oli erittäin helppo ohjelman helppokäyttöisyyden ja olemassa olevien lisäosien takia.

Työn toteutus oli mielenkiintoinen sekä onnistui omasta mielestäni hyvin. Ohjelmiston laatu ja suorituskyky on jo parantunut kehitystoimien johdosta. Automaatio varmistaa laadun pysyvyyden ja mittaustuloksen tarjoaa statistiikkaa laadunparannusmittareille.

Lähteet

1 Meteoriitti – Ketteryys haltuun. 2011. Verkkodokumentti. <http://www.meteoriitti.com/Artikkelisarjat/Ketteryys-haltuun/>. Luettu 28.11.2014.

2 Beck Kent, Agile Manifesto – Principles behind the Agile Manifesto. 2001. Verkkodokumentti. <http://www.agilemanifesto.org/principles.html>. Luettu 28.11.2014.

3 The Scrum Guide. 2014. Verkkodokumentti. <http://www.scrumguides.org/scrum-guide.html>. Luettu 5.10.2014.

4 Agile Alliance – Continuous Integration. Verkkodokumentti. <http://guide.agilealliance.org/guide/ci.html>. Luettu 28.11.2014.

5 Gofore – Automatisoitu ohjelmiston tuotantoprosessi. 2013. Verkkodokumentti. <http://gofore.com/ohjelmistokehitys/automatisoitu-ohjelmiston-tuotantoprosessi/>. Luettu 29.11.2014.

6 Tuomas Kautto – Ohjelmistotestaus ja siinä käytettävät työkalut. 1996. Verkko-dokumentti. <http://www.mit.jyu.fi/opiskelu/seminaarit/bak/testaus/#RTFToC7>. Luettu 5.12.2014.

7 Sikuli. 2014. Verkkodokumentti. <http://sikulix-2014.readthedocs.org/en/latest/index.html>. Luettu 7.10.2014.

8 Smartbear – TestComplete. 2014. Verkkodokumentti. <http://support.smartbear.com/viewarticle/63445/?st=0>. Luettu 7.10.2014.

9 PerfPublisher. 2014. Verkkodokumentti. <https://wiki.jenkins-ci.org/display/JENKINS/PerfPublisher+Plugin>. Luettu 15.10.2014.

10 PhantomJS. 2014. Verkkodokumentti. <http://phantomjs.org/documentation/>. Luettu 15.10.2014.

11 ImageMagick. 2014. Verkkodokumentti. <http://www.imagemagick.org/script/command-line-tools.php>. Luettu 15.10.2014.

12 Smartbear – TestExecute Command Line. 2014. Verkkodokumentti. <http://support.smartbear.com/viewarticle/54705/>. Luettu 15.10.2014.

13 Smartbear – AQTime. 2014. Verkkodokumentti. <http://support.smartbear.com/viewarticle/43079/?st=0>. Luettu 10.11.2014.

14 Andrew Yang – Leveraging static code analysis for medical device software. 2010. Verkkodokumentti. <http://embedded-computing.com/articles/leveraging-analysis-medical-device-software/ >. Luettu 4.11.2014.

15 Cppcheck 1.67. 2014. Verkkodokumentti. <http://cppcheck.sourceforge.net/manual.pdf>. Luettu 5.11.2014.